

## Objectives

- Design Patterns

Nov 14, 2011

Sprenkle - CSCI209

1

## Animation Review

- What type of object do we use to “draw” in Java?
  - What are some things we can do?

My German Word of the Day: die Benutzeroberfläche

Nov 14, 2011

Sprenkle - CSCI209

2

## DESIGN PATTERNS

Nov 14, 2011

Sprenkle - CSCI209

3

## Design Pattern

General reusable solution to a commonly occurring problem in software design

- Not a finished design that can be transformed directly into code
- Description or *template* for how to solve a problem that can be used in many different situations
  - “Experience reuse”, rather than code reuse

Nov 14, 2011

Sprenkle - CSCI209

4

## Defined Design Patterns

- Software best practices
- Catalogued and discussed in *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Written by the “Gang of Four”:  
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
    - Erich Gamma also co-wrote JUnit framework
  - Didn’t design the patterns; identified them

Nov 14, 2011

Sprenkle - CSCI209

5

## Applying Design Patterns

1. Recognize problem as one that can be solved by a design pattern
2. Apply pattern to your problem

**Danger:** over-applying design patterns

- Fall back: Identify and resolve code smells

Nov 14, 2011

Sprenkle - CSCI209

6

## Motivating Example

- Birds
  - Various flying behaviors (some fly, some don't)
  - Make different sounds
  - Examples: Duck, Penguin, Hummingbird, Ostrich, Chicken, Oriole, ...

How can we represent different birds?

Nov 14, 2011

Sprenkle - CSCI209

7

## Designing Flexible Behaviors

- Include behaviors in abstract Bird class
  - FlyBehavior object has performFly() method
  - SoundBehavior object has makeSound() method
- Could have setter methods in Bird class to change these
  - Example: bird's wings get clipped

Nov 14, 2011

Sprenkle - CSCI209

8

## Designing Flexible Behaviors

```
public abstract class Bird {
    protected FlyBehavior flyB;
    protected SoundBehavior soundB;

    public Bird() {
        ...
    }

    public void performSound() {
        soundB.makeSound();
    }

    public void performFly() {
        flyB.performFly();
    }
}
```

9

## Designing Flexible Behaviors

```
public class Duck {
    //Recall: protected FlyBehavior flyB;
    //Recall: protected SoundBehavior soundB;

    public Duck() {
        ...
    }
}
```

What do we need to do in here?

Nov 14, 2011

Sprenkle - CSCI209

10

## Designing Flexible Behaviors

```
public class Duck {
    public Duck() {
        flyB = new FlyHighBehavior();
        soundB = new QuackBehavior();
    }
}
```

Do we need to do anything else to *this* class, with respect to fly and sound behavior?

Nov 14, 2011

Sprenkle - CSCI209

11

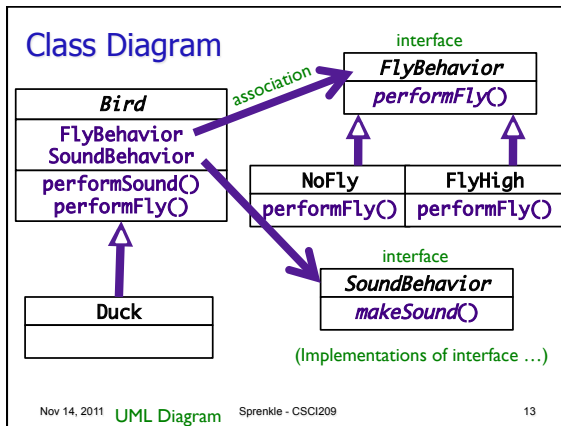
## How Do We Implement...

- Hummingbird?
- Penguin?
- Ostrich?

Nov 14, 2011

Sprenkle - CSCI209

12



## Unified Modeling Language (UML)

- Standardized general-purpose modeling language
  - Graphical language for visualizing, specifying and constructing the artifacts of a software system
- Includes a set of graphical notation techniques to create abstract models of specific systems
- Used in designing a large system
  - Focus on big picture, not the code

Nov 14, 2011

Sprenkle - CSCI209

14

### Design Principle: Favor Composition Over Inheritance

- Composition
  - Using other objects in your class
  - "Delegate" responsibilities to this object

Why is composition preferred over inheritance?

Nov 14, 2011

Sprenkle - CSCI209

15

### Design Principle: Favor Composition Over Inheritance

- Composition
  - Using other objects in your class
  - "Delegate" responsibilities to this object

Why is composition preferred over inheritance?

- Inheritance → dependence on parent class
  - Only want to depend on things you know won't change (higher stability)
- Composition: Provide different behaviors for your class by plugging in new object

Nov 14, 2011

Sprenkle - CSCI209

16

## Another Solution: Using Interfaces

- We could have a Flyable interface with a performFly() method and a Chirpable interface with a chirp() method
- Then, each bird class would implement Flyable and Chirpable, as appropriate

Pros and cons of this solution?

Nov 14, 2011

Sprenkle - CSCI209

17

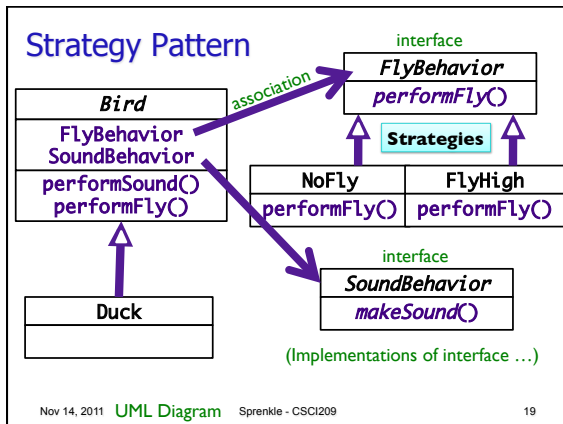
## Pros and Cons of Interface Solution

- We could have a Flyable interface with a performFly() method and a Chirpable interface with a chirp() method
- Pros: Using an interface → more flexible
  - Depending on interface instead of implementation
- Con: Duplicated code, implement in each class

Nov 14, 2011

Sprenkle - CSCI209

18



## Design Pattern: Strategy

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Lets algorithm/behavior vary independently from clients that use it
  - Allows behavior changes at runtime
- Design Principle:

Favor **composition** over inheritance

Nov 14, 2011

Sprenkle - CSCI209

20

## What Are the Benefits of the Strategy Pattern?

Nov 14, 2011

Sprenkle - CSCI209

21

## What Are the Benefits of the Strategy Pattern?

- Uses **delegation** ← Pattern in its own right
  - Reduces Bird's responsibilities
    - Delegate some responsibilities to SoundBehavior and FlyBehavior
  - Reduces Bird's code
- Easy swap of different strategy
  - Because have **one interface**, can easily plug in different behavior/implementation
    - Coding to interface, not implementation

Nov 14, 2011

Sprenkle - CSCI209

22

## Discussion: Applying Design Patterns

- When should we apply the **delegation** pattern?
  - Example, if X, then we should apply the pattern.
- When should we apply the **strategy** pattern?
- When will we know we've gone too far (overapplying)?
  - What are some symptoms to look for?

Nov 14, 2011

Sprenkle - CSCI209

23

## Discussion: Applying Design Patterns

- When should we apply the **delegation** pattern?
  - When we know that the requirements or implementations for a responsibility are likely to change
    - Change: Number/types of birds; types of behaviors; or lower-level implementation details
- When should we apply the **strategy** pattern?
  - When there are lots of desired behaviors for one responsibility
- When will we know we've gone too far (overapplying)? What are some symptoms to look for?
  - "Too small" classes → don't do anything
  - Have many more strategies than necessary
  - "Speculative generality"

Nov 14, 2011

Sprenkle - CSCI209

24

## Design Pattern: **Factory Methods**

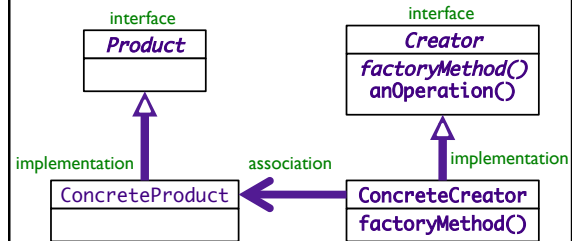
- Allows creating objects without specifying exact (concrete) class of created object
- Often used to refer to any method whose main purpose is creating objects
- How it works:
  1. Define a method for creating objects
  2. Child classes override method to specify the derived type of product that will be created

Nov 14, 2011

Sprenkle - CSCI209

25

## Factory Method Pattern



UML Class Diagram

Nov 14, 2011

Sprenkle - CSCI209

26

## Mapping Factory Design Pattern to Screen Savers

- How does the screen saver application use factory methods?
- What would be the alternative solution?
- What problems are the factories addressing?

Nov 14, 2011

Sprenkle - CSCI209

27

## Mapping Factory Design Pattern to Screen Savers

- How does the screen saver application use factory methods?
- What would be the alternative solution?
- What problems are the factories addressing?
  - Delegate creation of concrete Movers
    - Likely to change
    - Encapsulate change in factory
  - Using abstraction instead of specifying concrete classes
    - Reduces dependencies to concrete classes

Nov 14, 2011

Sprenkle - CSCI209

28

## Thoughts

- Didn't *need* to know design pattern to understand code
  - Helps to know the **terminology** to understand the naming
- Design principles all come down to *where there is change, use abstraction*

Nov 14, 2011

Sprenkle - CSCI209

29

## Dependency Inversion Principle

Depend upon abstractions.  
Do not depend upon concrete classes.

- High-level components should not depend on low-level components
  - Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions
- "Inversion" from the way you think
- Other techniques besides Factory Method for adhering to principle

Nov 14, 2011

Sprenkle - CSCI209

30

## Dependency Inversion Principle

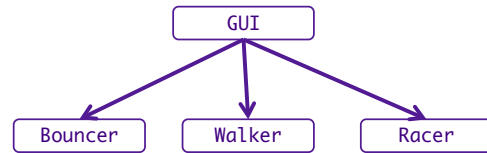
- How would we build/design the screen saver application?
  - Know we need to view/display a screen saver
    - Buttons, slider, objects that move
    - Top-down
  - Know we need to create a bunch of types of screen savers
    - Abstraction
    - Bottom-up

Nov 14, 2011

Sprenkle - CSCI209

31

## One Option for Screen Saver Dependencies



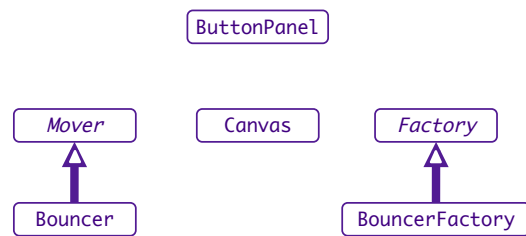
High-level component is dependent on concrete classes.  
If implementations change, GUI may have to change

Nov 14, 2011

Sprenkle - CSCI209

32

## Our Screen Saver Dependencies

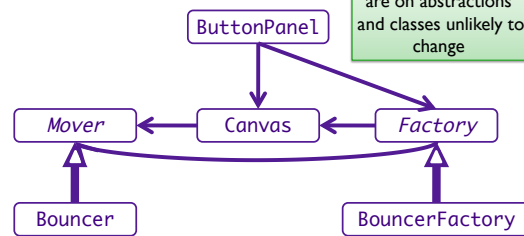


Nov 14, 2011

Sprenkle - CSCI209

33

## Screen Saver Dependencies



Note: dependencies are on abstractions and classes unlikely to change

Nov 14, 2011

Sprenkle - CSCI209

34

## Guidelines to Follow DIP

- No variable should hold a reference to a concrete class
  - Using `new` → holding reference to concrete class
  - Use factory instead
- No class should derive from a concrete class
  - Why? Depends on a concrete class
  - Derive from an interface or abstract class instead
- No method should override an implemented method of its base class
  - Base class wasn't an abstraction
  - Those methods are meant to be shared by subclasses

What's the problem with following all of these guidelines?

Nov 14, 2011

## Dependency Inversion Principle

Depend upon  
abstractions

Nov 14, 2011

Sprenkle - CSCI209

36

## To Do

- Assign 11: Screensavers due Friday
- Extra Credit: Naomi Oreskes talk, 5:30 Stackhouse
  - [Answer questions on Sakai](#)
    - 3 most important points of her talk
    - most surprising thing she mentioned
    - at least one question that you wondered during the talk
    - one problem that she posed that a computer scientist could help solve; tell me a little about your proposed solution

Nov 14, 2011

Sprenkle - CSCI209

37